CyCLaDEs: A Decentralized Cache for Triple Pattern Fragments

Pauline Folz[1,2], Hala Skaf-Molli[1], and Pascal Molli[1]

[1] LINA– Nantes University, France
{pauline.folz,hala.skaf,pascal.molli}@univ-nantes.fr
[2] Nantes Métropole - Research, Innovation and Graduate Education Department, France

Abstract. The Linked Data Fragment (LDF) approach promotes a new trade-off between performance and data availability for querying Linked Data. If data providers' HTTP caches plays a crucial role in LDF performances, LDF clients are also caching data during SPARQL query processing. Unfortunately, as these clients do not collaborate, they cannot take advantage of this large decentralized cache hosted by clients. In this paper, we propose CyCLaDEs an overlay network based on LDF fragments similarity. For each LDF client, CyCLaDEs builds a neighborhood of LDF clients hosting related fragments in their cache. During query processing, neighborhood cache is checked before requesting LDF server. Experimental results show that CyCLaDEs is able to handle a significant amount of LDF query processing and provide a more specialized cache on client-side.

1 Introduction

Following Linked Data principles, data providers made billions of triples available on the web [4] and the number of triples is still growing [14]. A part of these data is available through public SPARQL endpoints maintained by data providers. However, public SPARQL endpoints have an intrinsic problem of availability as observed in [1]. The Linked Data Fragments (LDF) [16] tackles this issue by balancing the cost of query processing between data providers and data consumers. In Linked Data Fragments, data are hosted in Linked Data Fragments (LDF) servers providing low-cost publication of data, at the same time, SPARQL query processing is moved to the LDF clients side. This approach establishes a trade-off between data availability and performances leveraging the "pressure" on data providers. Consequently, a data provider can provide many datasets through one LDF server at a low-cost as demonstrated in WarDrobe [2] where more than 657,000 datasets are provided with few LDF servers[3].

Caching plays an important role in the performance of LDF servers [17]. Client-side SPARQL query processing using Triple-Pattern Fragments (TPF) generates many calls to LDF server. But as queries are decomposed into triple

[3] http://lodlaundromat.org/wardrobe/

patterns, an important percentage of calls are intercepted by traditional HTTP caching techniques and leverage the pressure on LDF servers. However, HTTP caches are still on the charge of data providers and in the case of multiple datasets, the cache could be useless if a query does not belong to frequently accessed datasets.

During query processing, LDF clients are also caching data, a client replicates triple pattern fragments in its local cache. Unfortunately, as clients do not collaborate, they cannot take advantage of this large decentralized cache hosted by the clients. Building a decentralized cache on client-side has been already addressed by DHT-based approaches [9]. However, DHT-based approaches introduce high latency during the lookup of a content and can slow down the performance of the system. Behave [10] builds a behavioral cache for users browsing the web by exploiting similarities between browsing behaviors of users. Based on past navigation, the browser is directly connected to a fixed number of browsers with similar navigation profile. Consequently, a new requested URL could be checked in the neighborhood cache with a zero-latency connection. A behavior approach has not been applied in the context of the semantic web. Performing SPARQL queries and navigating on the web are different in terms of the number of HTTP calls per-second and clients profiling.

In this paper, we propose CyCLaDEs an approach that allows to build a behavioral decentralized cache hosted by LDF clients. More precisely, CyCLaDEs builds a behavioral decentralized cache based on Triple-Pattern Fragments (TPF). The main contributions of the paper are:

- We present CyCLaDEs an approach to build a behavioral decentralized cache on client-side. For each LDF client, CyCLaDEs builds a neighborhood of LDF clients hosting similar triple pattern fragments in their cache. A neighborhood cache is checked before requesting LDF server.
- We present an algorithm to compute clients profiles. The profile characterizes the content of the cache of LDF client at a given moment.
- We evaluate our approach by extending LDF client with CyCLaDEs. We experiment the extension in different setups, results show that CyCLaDEs reduces significantly the load on LDF server.

The paper is organized as follows: section 2 summarizes related works. Section 3 describes the general approach of CyCLaDEs. Section 4 defines CyCLaDEs model. Section 5 reports our experimental results. Finally, conclusions and future works are outlined in Section 6.

2 Related work

Improving SPARQL query processing with caching has been already addressed in the semantic web. Martin et al. [13] proposes to cache query results and manage cache replacement, Schmachtenberg [15] proposes a semantic query caching relying on queries similarity, and Hartig [11] proposes caching to improve efficiency and results completeness in link traversal query execution. All these approaches

rely on a temporal locality where specific data are supposed to be reused again with a relatively small time duration, and caching resources are provided by data providers. CyCLaDEs relies on behavioral locality where clients with similar profiles are directly connected, and caching resources are provided by data consumers.

The Linked Data Fragments (LDF) [17, 16] propose to shift complex query processing from servers to clients to improve availability and scalability of SPARQL endpoints. A SPARQL query is decomposed into triple patterns, an LDF server answers triple patterns and sends data back to the client. The client performs joins operations based on the nested loop operators, the triples patterns generated during the query processing are cached in the LDF client and in the traditional HTTP cache in front of the LDF Server. Although, a SPARQL query processing increases the number of HTTP requests to the server, a large number of requests are intercepted by the server cache reducing significantly the load on LDF server as demonstrated in [17]. LDF relies on a temporal locality, and the data providers have to provide resources for the data caching. Compared to other caching techniques in the semantic web, the LDF cache results of a triple pattern, increasing their usefulness for other queries, *i.e*, the probability of a cache hit is higher than the caching of a SPARQL query results. CyCLaDEs aims at discovering and connecting dynamically LDF clients according to their behaviors. CyCLaDEs makes the hypothesis that clients perform a limited query mix, consequently, a triple pattern of a query could be answered in a neighbor cache. To build a decentralized behavior cache, each LDF client must have a limited number of neighbors with a zero-latency access. During query processing, for each triple pattern subquery, CyCLaDEs checks if the triple pattern can be answered in the local cache, if not, in the cache of neighbors. A request is sent to LDF server only if the triple pattern cannot be answered neither in the local cache nor in the neighbors cache. CyCLaDEs improves LDF approach by hosting behavioral caching resources on the clients-side. Behavior cache reduces calls to an LDF server, especially, when the server hosts multiple datasets, the HTTP cache could handle frequent queries on a dataset but cannot absorb all calls. In other words, unpopular queries will not be cached in the HTTP cache and will be answered by the server. In CyCLaDEs, the neighborhood depends on fragments similarity which means that clients are gathered in communities depending on their past queries. By doing that unpopular or less frequent queries can be handled in the cache of the neighbors.

Decentralized cooperative caches were proposed in many research areas. Dahlin et al. [8] proposes a cooperative caching to improve file system read response time. By analyzing existing large-scale Distributed File Systems (DFS) workloads, Blaze [6] discovers that large proportion of "cache miss" is for files that are already copied in another client's cache. Blaze proposes dynamic hierarchical caching to reduce "cache miss" traffic for DFS and server's load. Research on peer-to-peer-oriented Content Delivery Networks (CDN) propose a decentralized web cache such as Squirrel [12], FlowerCDN [9] and Behave [10]. Squirrel and FlowerCDN use Distributed Hash Table (DHT) for indexing all content at all

peers. If such approaches are relevant, querying the cache is expensive in term of latency. With n participants, a DHT requires $log(n)$ access to check the presence of a key in the DHT. As LDF query processing can generate thousands of sub-calls, DHT latency becomes a bottleneck and querying the DHT is considerably less performant than querying directly the LDF server.

Behave [10] is a decentralized cache for browsing the web. It is based on the Gossple approach [3]. The basic hypothesis is: if two users had visited the same web page in the past, they will likely to exhibit more common interests in the future. Based on this assumption, Behave relies on gossiping techniques to build dynamically a fixed-size neighboring for clients based on their profile, *i.e.*, their past HTTP access. When requesting a new URL, Behave is able to quickly checks if this URL is available in neighborhood. Compared to the DHT, the number of items available in Behave cache is smaller, but they are available with zero-latency, *i.e.*, with a direct socket or web socket connection. The available items are also personalized, they are based on the behavior of the client rather than a temporal locality.

In CyCLaDEs, we want to apply the general approach of Behave for LDF clients. However, compared to the human browsing, an LDF client could process a large number of queries per second and the local cache of the client could change quickly. We make the hypothesis that the clients processed same queries in the past will likely process similar queries in the future. We build a similarity metric by counting the number of predicates in the triple patterns on a sliding window. We demonstrate that this metric is efficient for building a decentralized cache for LDF clients.

3 CyCLaDEs Motivation and Approach

In CyCLaDEs, we make the assumption that clients who processed same queries in the past will likely process similar queries in the future. This is the case of a web applications proposing forms to the end-users and then executes parametrized SPARQL queries. The Berlin SPARQL Benchmark (BSBM) is built like that [5]. BSBM supposes a realistic web application where the users can browse products and reviews. BSBM generates a query mix based on 12 queries template and 40 predicates.

CyCLaDEs aims to build a behavioral decentralized cache for LDF query processing based on the similarities of LDF clients profiles. For each client, CyCLaDEs selects a fixed number of the best similar clients called *neighbors* and establishes a direct network connections with them. During a query processing on a given client, each triple pattern subquery is checked first on the local cache, next in the cache of the neighbors, before contacting the LDF server, if necessary. Because CyCLaDEs adds a new verification in the neighborhood, checking the cache of neighbors quickly is essential for the performance of CyCLaDEs. We expect that the orthogonality of behavioral cache hosted by the data consumers, and the temporal cache hosted by the data providers will reduce significantly the load on the LDF servers.

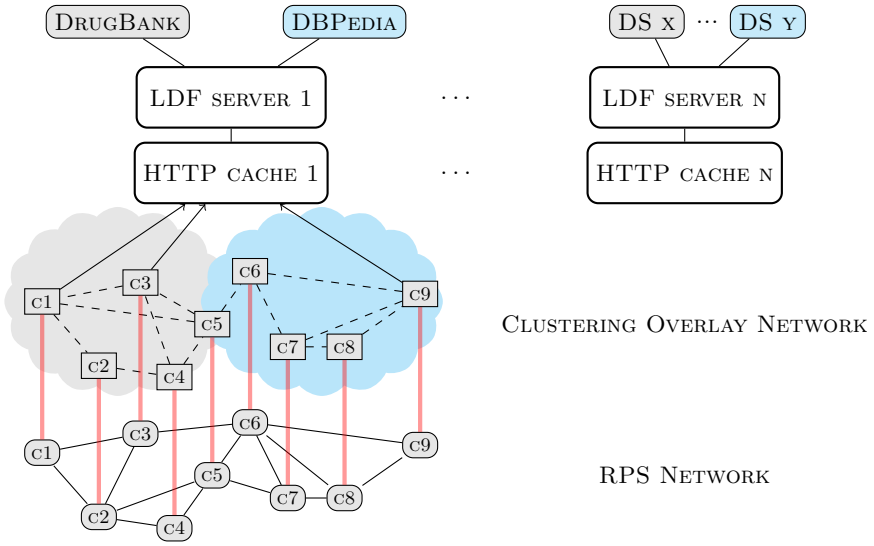


Fig. 3.1: c1-c9 represents LDF clients executing queries on LDF server 1. The RPS network connects clients in a random graph. CON network connects the same clients (red link) according to their queries. c1-c4 performs queries on DrugBank. c6-c9 perform queries on DBpedia. c5 performs queries on both. The total number of LDF servers is N.

In order to build a neighborhood and handle the dynamicity of the clients, we follow the general approach of Gossple [3]. CyCLaDEs builds two overlay networks on the client-side :

1. a *Random Peer Sampling (RPS)* overlay network that maintains the membership among connected clients. We rely on the Cyclon protocol [18] to maintain the network. Each client maintains a partial view on the entire network. The *view* contains a random subset of network nodes. Periodically, the client selects the oldest node from its view and they exchange parts of their views. This view is used to bootstrap and maintain the clustering network.
2. a *Clustering Overlay Network (CON)* builds on top of RPS, it clusters clients according to their profile. Each client maintains a second view, this view contains the *k-best* neighbors according to the similarity of their profile with the client profile. The maintenance of *k-best* neighbors is performed at RPS exchange time. To minimize the overhead of shuffling: (1) the profile informations have to be as small as possible, (2) the similarity metric has to be computed quickly in order to prevent slowing down the query engine.

Figure 3.1 shows LDF clients, clients $c1 - c4$ performs queries on DrugBank, $c6 - c9$ performs queries on DBpedia and $c5$ performs queries on both. The RPS network ensures that all clients are connected through a random graph, clients

profiles make the clustered network converging towards two communities. $c1-c4$ will be highly connected because they access data over DrugBank, while $c6-c9$ will be grouped together due their interest in DBpedia. $c5$ could be connected to both communities because it performs query on DrugBank and DBpedia.

Thanks to the clustered overlay network, a client is now able to check the availability of triple patterns in its neighborhood before sending the request to the HTTP cache. Under the hypothesis of profiled clients, the behavioral cache should be able to handle a significant number of triple pattern queries and to scale with the number of clients without requesting new resources from the data providers. Of course, the behavioral cache is efficient if the neighborhood of each client is pertinent and the overhead of the networks maintenance is still low.

4 CyCLaDEs model

In this section, we detail the model of the overlay networks built by CyCLaDEs on the client-side.

4.1 Random Peer Sampling

Random Peer Sampling (RPS) protocols [18] allow each client to maintain a view of a random subset of the network called *neighbors*. A view is a fixed-size table, associating a client ID to an IP address. The size of this view can be set to $log(N)$, where N is the total number of the node in the network. RPS protocols ensure that the network converges quickly to a random graph with no partitions, *i.e.*, a connected graph.

To maintain the connectivity of the overlay network, a client periodically selects the oldest node from its view and they exchange parts of theirs views. These periodic shuffling of the views of the clients ensures that each client view always contains a random subset of the network nodes and consequently maintains the clients connected through a random graph.

In CyCLaDEs, to ease the joining of the network, LDF server maintains a list of three last connected clients, *i.e*, called *bootstrap clients*. Each time a new client joins the network, *i.e.*, contacts the LDF server, the client receives automatically a list of the three last connected clients and add randomly one of them in its view. Periodic shuffling quickly re-establish the random graph property on the network including the new client.

4.2 LDF client Profiles

The Clustering Overlay Network (CON) relies on the LDF client profile. A client profile has to characterize the content of the local cache of a client. At a given instant, the content of the cache is determined by the result of recent past processed queries.

The cache of a LDF client is a list of $(key, value)$ fixed-size LRU cache. The key is a triple pattern fragment where the predicate is a constant and the $value$

is the set of triples that matches the fragment [16]. Each fragment matching a triple pattern fragment is divided into pages, each page contains 100 triples. The fragment is filled asynchronously and can be "incomplete", *e.g.*, a fragment $f1$ matches 1,000 triples, but currently only the first 100 triples has been retrieved from the server.

To illustrate, suppose that a LDF client is processing the following SPARQL query:

```
SELECT DISTINCT ?book ?author
WHERE {
  ?book rdf:type dbpedia-owl:Book;          tp1
        dbpedia-owl:author ?author.         tp2
}
LIMIT 5
```

Listing 1.1: Q: Authors of books

The query is decomposed into triple pattern $tp1$ and $tp2$. The local cache will be asynchronously populated as described in table 1. Because the number of matches of books (31,172) is smaller than those of authors (39,935), LDF client starts by retrieving books. Entry 0 contains $tp1$ with empty data, entry 1 contains $tp2$ with some data. LDF client starts by retrieving books and starts the nested loop to retrieve authors for a given book. Entries 2-9 contain all one triple as answer, but only the first five are needed to be retrieved to answer the query with *Limit* 5. Several strategies are possible to compute a profile of a LDF client:

- *Cache key*: we can consider a vector of keys of the cache as in Behave [10] and we reduce the dimension of the vector with a bloom filter. However, nested loop processing makes LDF quickly override the whole cache with the next query. If a new query searching for French authors is executed, then the nested loop will iterate on French authors instead of books and will completely rewrite the cache given in table 1. Consequently, the state of the cache at a given time, do not reflect the near past.
- *Past queries*: we can analyze statically the past executed queries and extract processed predicates. Unfortunately, this does reflect the join ordering decided at run-time by LDF client and cannot take into account nested loops.
- *count-min sketch*: we can use the count-min sketch [7] to analyze the frequency of processed predicates from the beginning of the session. However, count-min sketch does not forget and will not capture the recent past.

In CyCLaDEs, we want to define the profile in spirit of *count-min sketch* but with a short time memory, *i.e.*, a memory of the recent past. We denote the profile of a client c by $Pr(c) = \{(p, f_p, t)\}$, where Pr is a view of a fixed-size on the stream of the triple patterns processed by the client, p is a predicate in the triple pattern in the stream, f_q is the frequency and t is the timestamp of the last update of p. To avoid to mix predicates retrieved from different data sources, we concat the predicate and the provenance of predicate. For example, the general

Table 1: LDF client cache after execution of Query in Listing 1.1

	key	triples
0	?book http:.../ontology/author ?author	[]
1	'?book' http://www.w3.org/1999/02/22-rdf-syntax-ns#type http:.../ontology/Book	http:.../resource/%22...And_Ladies_of_the_Club%22 http://www.w3.org/1999/02/22-rdf-syntax-ns#type http:.../ontology/Book ... http:.../resource/%22K%22_Is_for_Killer http://www.w3.org/1999/02/22-rdf-syntax-ns#type http:.../ontology/Book
2	http:.../resource/%22...And_Ladies_of_the_Club%22 http:.../ontology/author ?author	http:.../resource/%22...And_Ladies_of_the_Club%22, http:.../ontology/author, http:.../resource/Helen_Hooven_Santmyer
3	http:.../resource/%22A%22_Is_for_Alibi http:.../ontology/author ?author	http:.../resource/%22A%22_Is_for_Alibi, http:.../ontology/author, http:.../resource/Sue_Grafton
4	http:.../resource/%22B%22_Is_for_Burglar, http:.../ontology/author, ?author	http:.../resource/%22B%22_Is_for_Burglar, http:.../ontology/author, http:.../resource/Sue_Grafton
5	http:.../resource/%22C%22_Is_for_Corpse, http:.../ontology/author, ?author	http:.../resource/%22C%22_Is_for_Corpse, http:.../ontology/author, http:.../resource/Sue_Grafton
6	http:.../resource/%22D%22_Is_for_Deadbeat, http:.../ontology/author, ?author	http:.../resource/%22D%22_Is_for_Deadbeat, http:.../ontology/author, http:.../resource/Sue_Grafton
7	http:.../resource/%22E%22_Is_for_Evidence, http:.../ontology/author, ?author	[]
8	http:.../resource/%22F%22_Is_for_Fugitive, http:.../ontology/author, ?author	[]
9	http:.../resource/%22G%22_Is_for_Gumshoe, http:.../ontology/author, ?author	[]

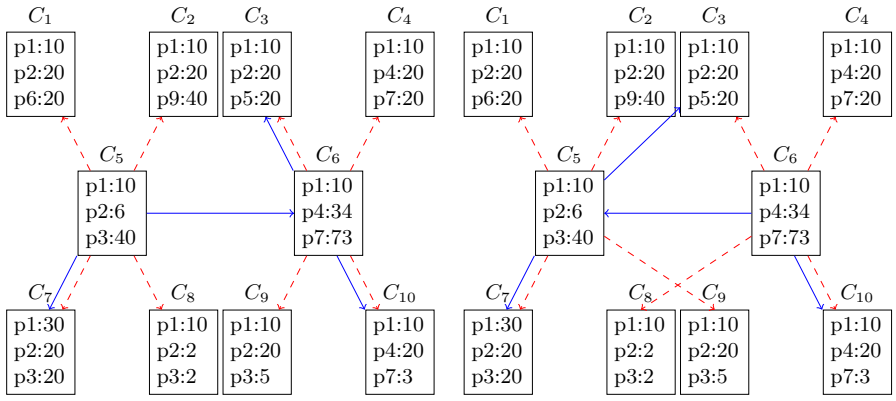
Algorithm 1 ComputeProfile(s,w,t)

Require: : w: Window size, s: Stream of triples, t: timestamp
Ensure: : Pr : set of (predicate, frequency, timestamp) of size w
1: Pr $\rightarrow \varnothing$
2: **while** data stream continues **do**
3: Receive the next streaming triple $tp = (s\ p\ o)$
4: **if** $(tp.p, f_p, \_) \in$ Pr **then**
5: $Pr.update(tp.p,\ f_p + 1, t)$ {accumulate the frequency of the predicate p and update time}
6: **else**
7: $Pr \cup (tp.p, 1, t)$ {add the new predicate p to the profile}
8: **if** $|Pr| >$ w **then**
9: $Pr \setminus (p_1, f_{p_1}, t_1) : (p_1, f_{p_1}, t_1) \in Pr \wedge \nexists\ (p_2, f_{p_2}, t_2) \in Pr : t_2 < t_1\}$ {delete the oldest predicate from the profile}

predicate *rdfs:label* retrieved from DBpedia should not be used with the same predicate retrieved from DrugBank. In order to simplify notation, we just keep *predicate* that should be expanded to the couple *(provenance,predicate)*.

The algorithm 1 presents CyCLaDEs profiling procedure. CyCLaDEs intercepts the stream of the processed triples and extracts the predicates. If the predicate belongs to the profile, the frequency of this predicate is incremented by one and the timestamp associated to this entry is updated. Otherwise, CyCLaDEs just insert a new entry in the profile. If the structure exceeds w entries, then CyCLaDEs removes the entry with the oldest timestamp. This profiling algorithm is designed to tolerate nested loops and forget predicates which are not used frequently. For the client whose cache is detailed in table 1, after the entry 4 in the cache, the profile will be:

{(`http://www.w3.org/1999/02/22-rdf-syntax-ns#type` , 1) ,
(`http://dbpedia.org/ontology/author` , 3)}



(a) C_5 starts shuffling with C_6

(b) State after shuffling.

Fig. 4.1: Partial CyCLaDEs network centred on C_5, C_6. Solid lines represent clients in RPS view (2). Dashed lines represent clients in CON view (4). Each client has a profile size of 3 defined as $predicate : frequency$.

4.3 Clustered Network and Similarity Metric

CyCLaDEs relies on a random peer sampling overlay network for managing memberships and on a clustered overlay network to manage the *k-best* neighbors. Concretely, the clustered network is just a second view on the network hosted by each client. This view is composed of the list of *k-best* neighbors with similar profiles. The view is updated during shuffling phase, when a client starts shuffling, it selects the oldest neighbor in its RPS view and they exchange profile informations, if the remote client has better neighbors in its view, then the local view is updated in order to keep the *k-best* neighbors.

To determine if a profile is better than another one, we use the generalized Jaccard similarity coefficient defined as:

$$J(x, y) = \frac{\sum_i min(x_i, y_i)}{\sum_i max(x_i, y_i)}$$

Where x and y are two multi-sets and the natural numbers $x_i \geqslant 0$ and $y_i \geqslant 0$ are the multiplicity of item i in each multiset.

Figure 4.1 describes a CyCLaDEs network focused on C_5, C_6. The RPS view size is fixed to 2 and is represented as solid lines. The CON view size is fixed to 4 and is represented as dashed lines. Each client has a profile of size 3 that contains the last 3 mostly used predicates in the recent past. p_i represents a predicate and the associated integer indicates the frequency of this predicate in the recent past. Figure 4.1a illustrates the state of C_5, C_6 before C_5 triggered a shuffling with C_6. C_6 is chosen because it is the oldest client in RPS view of C_5. Figure 4.1b describes the state of C_5, C_6 after completion of shuffling. As

we can see, only one RPS neighbor is changed for both C_5 and C_6. This is the result of exchanging half of RPS view between C_5 and C_6 as in Cyclon [18]. For CON views, C_5 integrated C_9 in its cluster while C_6 integrated C_8. During shuffling, C_5 retrieves the profiles of the CON view of C_6 including C_6. Next, it ranks all profiles according to the generalized Jaccard coefficient and keeps only the top-4. C_9 is more similar to C_5 than C_8 because $J(C_5, C_9) = 0,3$, and $J(C_5, C_8) = 0,25$ therefore, C_5 drops C_8 and integrates C_9. C_6 follows the same procedure by dropping C_9 and integrating C_8.

5 Experimental Study

The goal of the experimental study is to evaluate the effectiveness of CyCLaDEs. We measure mainly the *hit-ratio*; the fraction of queries answered by the decentralized cache.

5.1 Experimental setup

We extended the LDF client[4] with the CyCLaDEs model presented in section 4. CyCLaDEs source code is available at : `https://github.com/pfolz/cyclades` [5]. The setup environment is composed of an LDF server, a reverse proxy and different number of clients. Nginx is the reverse proxy with a cache set to 1GB. We used Berlin SPARQL Benchmark (BSBM) [5] as in [16] with two datasets: 1M and 10M. We randomly generated 100 different query mix of the "explore" use-case of BSBM. Each query mix is composed of 25 queries and each client has its own query mix.

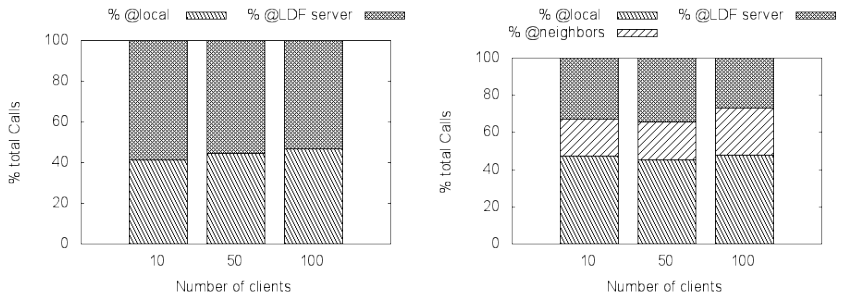
Table 2: Experimental Parameters

Parameter	Values
Number of Clients	10 - 50 - 100
RPS view	4 - 6 - 7
CON view	9 - 15 - 20
Local cache	100 - 1000 - 10000
Profile size	5 - 10 - 30
Shuffle Time	10s
Data sets	BSBM 1M - BSBM 10M
Queries	25 over BSBM

Table 2 presents the different parameters used in the experiment. We vary the value of parameters according to the objective of the experimentations as

[4] https://github.com/LinkedDataFragments/Client.js

[5] The current implementation does not handle the introduction to the network and fragment transfer, *i.e*, data are retrieved from the LDF server eventually.



(a) Original LDF client (b) LDF client with CyCLaDEs

Fig. 5.1: Impacts of clients number on hit-rate : (10 clients, $RPS_{view} = 4$, $CON_{view} = 9$), (50 clients, $RPS_{view} = 6$, $CON_{view} = 15$) and (100 clients, $RPS_{view} = 7$, $CON_{view} = 20$).

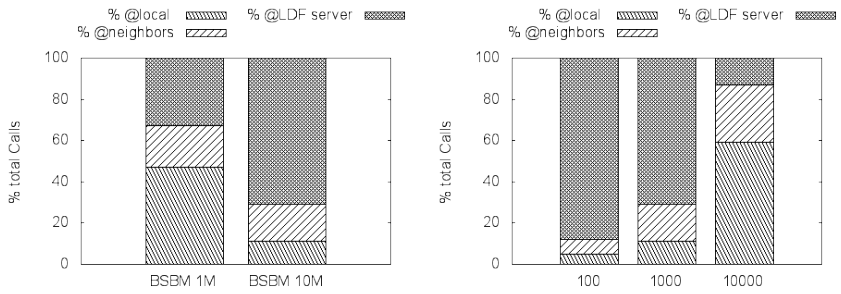
explained in the following sections. The shuffle time is fixed to 10s for all experiments. For all experimentations, we first run a warm-up round and start the real round after a synchronization barrier. The warm-up round bootstraps the network, local cache and HTTP cache. In both rounds, each client executes its own query mix of 25 queries in the same order. Hit-ratio is measured during the real round.

Impact of the number of the clients on the behavioral cache To study the impact of the number of clients on the behavior cache, we used BSBM dataset with 1M with a local cache of size 1,000 on each client. The RPS view size and CON view size are fixed to (4,9) for 10 clients, (6,15) for 50 clients, and (7,20) for 100 clients.

Figure 5.1a presents results of the LDF clients without CyCLaDEs. As we can see, ≈40% of calls are handled by the local cache, regardless the number of clients . The flow of BSBM queries simulates a real user interacting with a web application. This behavior promotes the local cache.

Figure 5.1b describes the results obtained with CyCLaDEs activated. The performances of local cache is nearly the same ≈40% of calls are handled by the local cache. However, ≈22% of total calls are answered in the neighborhood, consequently, the number of calls to the LDF server are considerably reduced. Moreover, for 100 clients number of calls answered by the server and the neighborhood are nearly the same. Because the CON view size for 100 clients is larger than those of 10 or 50 clients.

Impact of the size of the data sets on the behavioral cache For this experimentation, we used two datasets, BSBM with 1M triples and BSBM with 10M triples, a local cache of 1,000, a profile view of size 10 and 10 LDF clients. $RPS_{view} = 4$ and $CON_{view} = 9$, as in previous experiment. Figure 5.2a shows



(a) Data sets size (b) Local cache size

Fig. 5.2: Impacts of data sets size and local cache size on hit-rate. For 10 LDF clients with $RPS_{view} = 4$, $CON_{view} = 9$ and $Profile_{view} = 10$.

the percentage of calls answered in the local cache, neighbors caches and in the LDF server using the two datasets. As we can see, the calls to the local cache depends considerably on the size of the data, the percentage of hit-rate is 47% in the case of BSBM with 1M, and it decreased to 11% for BSBM with 10M. This is normal because the cache has a limited size and the temporal locality of the cache reduce its utility. However, the behavior cache calls stay stable with a hit-rate around 19% for both data sets.

Impact of the cache size We study the impact of the local cache size on the hit-rate of behavioral cache. We used the following parameters: BSBM 10M, 10 LDF clients, and $RPS_{view} = 4$ and $CON_{view} = 9$. Figure 5.2b shows that the number of calls answered by caches are proportional with the size of the cache. For local cache with 100 entries, the hit-rate of local cache and behavioral cache nearly equivalent 5% for the local cache and 7% for the behavioral cache. For local cache with 1,000 entries, the hit-rate behavioral cache is 18%, greater than the hit-rate of the local cache of 11%. Behavioral cache is more efficient than local cache. The situation changes for a local cache with 10,000 entries, in this case, the hit-rate of local cache is 59% and 28% for behavioral cache, only 13% of calls are forwarded to the server.

Impact of the profile size on cache-hit We run an experimentation with 2 different BSBM datasets of 1M, hosted on the same LDF server with 2 differents URLs. Each dataset has its own community of 50 clients running BSBM queries. As pointed out in section 4.2, we use provenance to differentiate predicates in local cache of LDF clients. All clients run with $RPS_{view} = 6$, $CON_{view} = 15$ and a cache size of 1,000 entries.

We vary profile size to 5, 10 and 30 predicates. Figure 5.3 shows that performances of CyCLaDEs are quite similar. However, the performances with $profile_{size} = 5$ is less than $profile_{size} = 10$ or 30. The query mix of BSBM use

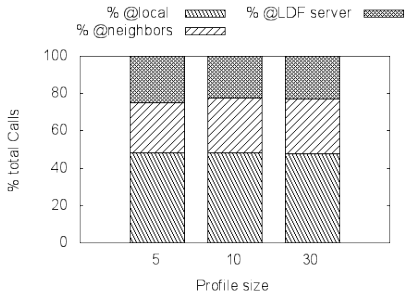


Fig. 5.3: Impacts of profile size on hit-rate for two datasets with 50 clients per dataset. $RPS_{view} = 6$ and $CON_{view} = 15$. $Profile_{size} = 5, 10$ and 30

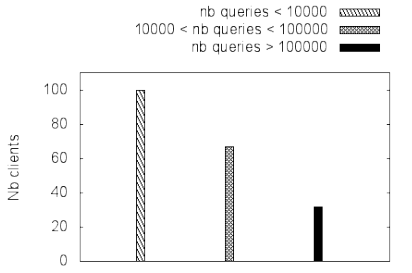


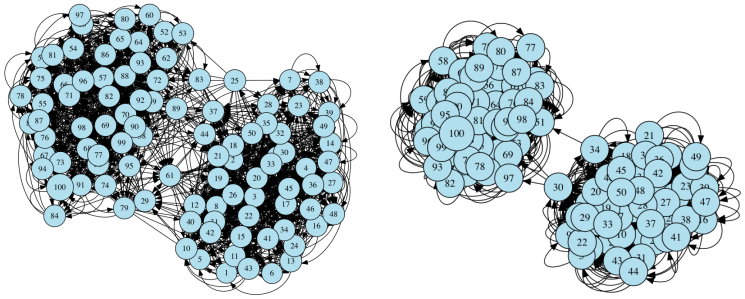
Fig. 5.4: Query distribution over clients

often 16 predicates. Therefore, 5 entries in the profile is sometimes not enough to compute a good similarity.

Query load As in the previous experimentation, we run a new experimentation with 2 different BSBM datasets of 1M hosted on the same LDF server with 2 different URLs.

Figure 5.4 shows the distribution of queries over clients. We want to verify if there is a hotspot, *i.e.*, one client receiving many cache queries from the others. As we can see, most of the clients handle 10,000 caches queries and a few handle more than 100,000 cache queries.

Impacts of the profile size on the communities in the clustering overlay network As in the previous experimentations, we set up 2 BSBM datasets of 1M with 50 clients per dataset. We vary the profile size to 5, 10 and 30. In the Figure 5.5, the directed graph represents the clustering overlay network where a node represents a client in the network and an edge represents the connection between clients. For example, an edge $1 \rightarrow 2$ means that the *client 1* has the *client 2* in its CON view. Figure 5.5 shows clearly that CyCLaDEs is able to build two clusters for both values of profile size. As we can see in Figure 5.5b, a



(a) Profile size = 5 (b) Profile size = 30

Fig. 5.5: Impacts of data profile size on the similarity in Clustering Overlay Network. Two distinct communities are discovered for two datasets.

greater value of profile size promotes the clustering, *i.e.*, only clients with similar profiles will receive queries to retrieve fragments.

6 Conclusion and Future work

In this paper, we presented CyCLaDEs, a behavioral decentralized cache for LDF clients. This cache is hosted by clients and completes the traditional HTTP temporal cache hosted by data providers.

Experimental results demonstrate that a behavioral cache is able to capture a significant part of triple pattern fragment queries generated by LDF query processing, in the context of web applications as described in section 3. Consequently, it leverages the pressure on data providers resources, by spreading the cost of query processing on clients. We proposed a cheap algorithm able to profile the subqueries processed on a client and gather the best neighbors for a client. This profiling has been proven effective in experiments. In this paper, we demonstrated how to bring data to queries with caching techniques, another approach could be to bring queries to data by choosing among neighbors, if a neighbor is able to process more than one triple pattern of a query. The promising results we obtained during experimentations encourage us to propose and experiment new profiling techniques that take into account the number of transferred triples and compare with the current profiling technique.

References

1. C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *12th International Semantic Web Conference (ISWC 2013)*, volume 8219 of *LNCS*, pages 277–293. Springer, 2013.
2. W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, and S. Schlobach. Lod laundromat: a uniform way of publishing other people's dirty data. In *13th International Semantic Web Conference (ISWC 2014)*, pages 213–228. Springer, 2014.

3. M. Bertier, D. Frey, R. Guerraoui, A. Kermarrec, and V. Leroy. The gossple anonymous social network. In *11th International Middleware Conference 'Middleware 2010) - ACM/IFIP/USENIX*, volume 6452 of *LNCS*, pages 191–211. Springer, 2010.
4. C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal of Semantic Web and Information Syststems*, 5(3):1–22, 2009.
5. C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009.
6. M. A. Blaze. *Caching in Large-scale Distributed File Systems*. PhD thesis, Princeton University, Princeton, NJ, USA, 1993. UMI Order No. GAX93-11182.
7. G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
8. M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *1st USENIX Conference on Operating Systems Design and Implementation (OSDI 1994)*, Berkeley, CA, USA, 1994.
9. M. El Dick, E. Pacitti, and B. Kemme. Flower-cdn: A hybrid p2p overlay for efficient query processing in cdn. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 427–438, New York, NY, USA, 2009. ACM.
10. D. Frey, M. Goessens, and A. Kermarrec. Behave: Behavioral cache for web content. In *Distributed Applications and Interoperable Systems (DAIS 2014)*, volume 8460 of *LNCS*, pages 89–103. Springer, 2014.
11. O. Hartig. How caching improves efficiency and result completeness for querying linked data. In *WWW2011 Workshop on Linked Data on the Web, Hyderabad, India, March 29, 2011*, 2011.
12. S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Twenty-first Annual Symposium on Principles of Distributed Computing (PODC 2002)*, pages 213–222, New York, NY, USA, 2002. ACM.
13. M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with sparql query caching. In *Extended Semantic Web Conference (ESWC 2010)*, pages 304–318, Berlin, Heidelberg, 2010. Springer-Verlag.
14. M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *13th International Semantic Web Conference (ISWC 2014)*, pages 245–260, 2014.
15. H. Stuckenschmidt. Similarity-based query caching. In *Flexible Query Answering Systems*, volume 3055 of *Lecture Notes in Computer Science*, pages 295–306. Springer Berlin Heidelberg, 2004.
16. R. Verborgh, O. Hartig, B. D. Meester, G. Haesendonck, L. D. Vocht, M. V. Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. V. de Walle. Querying Datasets on the Web with High Availability. In *13th International Semantic Web Conference (ISWC 2014)*, pages 180–196, 2014.
17. R. Verborgh, M. V. Sande, P. Colpaert, S. Coppens, E. Mannens, and R. V. de Walle. Web-Scale Querying through Linked Data Fragments. In *WWW Workshop on LDOW 2014*, 2014.
18. S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.